Miscellany

L4 reading: Virtual write queue by Stuechli et al

Assignment 1 due today
• 2 late days w/out penalty for semester
• Automatically taken care of by Autolab

If you submitted assignment 1 and aren’t enrolled, contact me!
Today and next: Parallel architecture

Different schools of parallel architecture
• I.e., programs are written to expose parallelism
• History of unconventional parallel architectures
• Convergence to today’s multiprocessor systems

We will learn...
• Why parallelism?
• Different models for parallel execution + associated architectures
• New challenges (communication, scalability) introduced by parallelism
Why parallelism?

For any given processing element, in principle:
more processing elements $\Rightarrow$ more performance

High-level challenges:
• Parallelism limited by communication barriers
e.g., cost between chips $\gg$ cost within a core
• Leads to design tradeoffs
e.g., $N$ small cores in 1 chip vs.
$N$ big cores in $N$ chips
• $N$ processors often $\neq N \times$ better performance
• Parallel programming is often hard
• $\Rightarrow$ What type of parallelism does app exploit best?
  • (In practice, machines exploit parallelism at multiple levels)
One Definition of Parallel Architecture

A parallel computer is a collection of processing elements that cooperate to solve large problems fast.

Key issues:

- **Resource Allocation**:
  - how large a collection?
  - how powerful are the elements?
  - how much memory?

- **Data access, Communication and Synchronization**
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?

- **Performance and Scalability**
  - how does it all translate into performance?
  - how does it scale?
Why Study Parallel Architecture and Programming?

The Answer from ~15 Years Ago:
- Mostly, Because it allows you to achieve performance beyond what we get with CPU clock frequency scaling
  - +30% freq/yr vs +40% transistors/yr—10× advantage over 20 yrs
- Rarely, Exploit concurrency for programmability

The Answer Today:
- Because it seems to be the best available way to achieve higher performance in the foreseeable future
- CPU clock rates are no longer increasing!
- Instruction-level-parallelism is not increasing either!
- Without explicit parallelism or architectural specialization, performance becomes a zero-sum game.
- Specialization is more disruptive than parallel programming
- Parallel architectures can simplify the architecture
- And reduce power (remember: $P = \frac{1}{2}CV^2F$ and $V \propto F \to P \propto CF^3$)
Recurring argument: technology is running out of steam; parallel architectures are more efficient than sequential processors (in perf/mm^2, power, etc)

Except that...

• ...technology defies expectations (at least until recently, eg, in ~early 2000s power became a 1st-order concern)

• ...parallelism is more efficient in theory, but getting good parallel programs in practice is hard (architecture doesn't exist in a vacuum; see also scratchpads vs caches)

⇒ Sequential/implicitly parallel arch dominant until ~15y ago
Historically, parallel architectures tied to programming models

- Divergent architectures, with no predictable pattern of growth.

Uncertainty of direction paralyzed parallel software development!
(Parallel programming remains a big problem)
Is parallel architecture enough?

NO. Parallel architectures rely on programs for performance.

AMBER code for CRAY-1; ported to Intel Paragon

(slide credit: Culler’99)
Types of Parallelism

Bit-level parallelism

- **Apply the same operation to many bits at once**
- 4004 4b $\Rightarrow$ 8008 8b $\Rightarrow$ 8086 16b $\Rightarrow$ 80386 32b
- E.g., in 8086, adding two 32b numbers takes 2 instructions (add, adc) and multiplies are 4 (mul, mul, add, adc)
- Early machines used transistors to widen datapath
- 32b $\Rightarrow$ 64b mostly not for performance, instead...
  - Floating point precision
  - Memory addressing (more than 4GB)
- *Not what people usually mean by parallel architecture today!*
Types of Parallelism

Instruction Level Parallelism

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

A:  
LD R2, 0(R1)  
LD R3, 4(R1)  
SUBI R2, R2, #1  
SUBI R3, R3, #1  
BLTZ R2, B  
ST R2, 0(R1)

B:  
BLTZ R3, C  
ST R3, 4(R1)

C:  
ADDI R1, R1, #8  
SUB R5, R4, R1  
BGTZ R5, A  
RET

void decrement_all(
    int *array,
    int size) {
    for (int i = 0;
        i < size;
        i++) {
        int x = array[i] - 1;
        if (x >= 0) {
            array[i] = x;
        }
    }
}

Loop unrolled x2
Types of Parallelism

Instruction Level Parallelism

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

A:  
LD R2, 0(R1)  
LD R3, 4(R1)  
SUBI R2, R2, #1  
SUBI R3, R3, #1  
BLTZ R2, B  
ST R2, 0(R1)  
B:  
BLTZ R3, C  
ST R3, 4(R1)  
C:  
ADDI R1, R1, #8  
SUB R5, R4, R1  
BGTZ R4, A  
RET

void decrement_all(int *array, int size) {
    for (int i = 0; i < size; i++) {
        int x = array[i] - 1;
        if (x > 0) {
            array[i] = x;
        }
    }
}

Loop unrolled x2
Types of Parallelism

Instruction Level Parallelism

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

A: 
- LD R2, 0(R1)
- LD R3, 4(R1)
- SUBI R2, R2, #1
- SUBI R3, R3, #1
- BLTZ R2, B
- ST R2, 0(R1)

B: 
- BLTZ R3, C
- ST R3, 4(R1)

C: 
- ADDI R1, R1, #8
- SUB R5, R4, R1
- BGTZ R4, A
- RET

LD

R1 += 8

R5=R4-R1

yes

no

>0?

>0?

yes

no

>0?

yes

no

ST

ST

RET

Start

CS 740 S'17
Types of Parallelism

Instruction-level parallelism peaks @ ~4 ins / cycle

Real programs w realistic cache+pipeline latencies, but unlimited resources
Types of Parallelism

Data Parallelism

- Different pieces of data can be operated on in parallel
- Vector processing, array processing
- Systolic arrays, streaming processors

(Not valid assembly)

```
MOV VLR, #2
A:
   CVM
   LV V1, 0(R1)
   SUBV V1, #1
   SLTV V1, #0
   SV V1, 0(R1)
   ADDI R1, R1, #8
   SUB R5, R4, R1
   BGTZ R5, A
   RET

R1 += 8
R5=R4-R1
>0? yes no
yes
no
>0? yes no
RET
```
Types of Parallelism

Data Parallelism

- Different pieces of data can be operated on in parallel
- Vector processing, array processing
- Systolic arrays, streaming processors

(Not valid assembly)

LUI VLR, #4
LV V1, 0(R1)
SUBV V1, #1
CLTV V1, #0
SV V1, 0(R1)
ADDI R1, R1, #16
SUB R5, R4, R1
BGTZ R5, A
RET
Types of Parallelism

Task Level Parallelism

- Different “tasks/threads” can be executed in parallel
- Multithreading
- Multiprocessing (multi-core)

Adjust R1, R5 per thread...

A:  
LD R2, 0(R1)  
SUBI R2, #1  
BLTZ R2, #0  
ST R2, 0(R1)  
ADDI R1, R1, #4  
SUB R5, R4, R1  
BGTZ R4, A  
RET
Types of Parallelism

Instruction Level Parallelism

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

Data Parallelism

- Different pieces of data can be operated on in parallel
- SIMD: Vector processing, array processing
- Systolic arrays, streaming processors

Task Level Parallelism

- Different “tasks/threads” can be executed in parallel
- Multithreading
- Multiprocessing (multi-core)
Flynn’s Taxonomy of Computers

Mike Flynn, “Very High-Speed Computing Systems,” 66

**SISD:** Single instruction operates on single data element

**SIMD:** Single instr operates on multiple data elements
- Array processor
- Vector processor

**MISD:** Multiple instrs operate on single data element
- Closest form?: systolic array processor, streaming processor

**MIMD:** Multiple instructions operate on multiple data elements (multiple instruction streams)
- Multiprocessor
- Multithreaded processor
Parallel architecture

Extension of “computer architecture” to support communication and cooperation

- OLD: Instruction Set Architecture
- NEW: Communication Architecture

Defines

- Critical abstractions, boundaries, and primitives (interfaces)
- Organizational structures that implement interfaces (hw or sw)

Compilers, libraries and OS are crucial bridges

Convergence crosses parallel architectures to include what historically were distributed systems.
Concurrent Systems

Embedded-Physical Distributed

Claytronics

Sensor Networks

Geographically Distributed
Concurrent Systems

Embedded-Physical Distributed

Claytronics

Sensor Networks

Geographically Distributed

Internet

Power Grid
**Concurrent Systems**

**Embedded-Physical Distributed**

Claytronics

Sensor Networks

**Geographically Distributed**

Internet

Power Grid

**Cloud Computing**

EC2

Azure
Concurrent Systems

Embedded-Physical Distributed

Claytronics

Sensor Networks

Geographically Distributed

Internet

Power Grid

Cloud Computing

Parallel

EC2

Azure
Modern Layered Framework

<table>
<thead>
<tr>
<th>CAD</th>
<th>Database</th>
<th>Scientific modeling</th>
<th>Parallel applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprogramming</td>
<td>Shared address</td>
<td>Message passing</td>
<td>Data parallel</td>
</tr>
<tr>
<td>Compilation or library</td>
<td>Operating systems support</td>
<td>Communication abstraction</td>
<td></td>
</tr>
<tr>
<td>Communication hardware</td>
<td>Hardware/software boundary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical communication medium</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Programming Model

What programmer uses in coding applications

Specifies operations, naming, and ordering – focus on communication and synchronization

Examples:

- Multiprogramming: no communication or synch. at program level
- Shared address space: like bulletin board, need separate synchronization (e.g., atomic operations)
- Message passing: like letters or phone calls, explicit point-to-point messages act as both communication and synchronization
- Data parallel: more regimented, global actions on data
  - Implemented today with shared address space or message passing
Communication Abstraction

User level communication primitives provided
- Realizes the programming model
- Mapping exists between language primitives of programming model and these primitives

Supported directly by hw, or via OS, or via user sw

Lot of debate about what to support in sw and gap between layers

Today:
- Hw/sw interface tends to be flat, i.e. complexity roughly uniform
- Compilers and software play important roles as bridges today
- Technology trends exert strong influence

Result is convergence in organizational structure
- Relatively simple, general purpose communication primitives
Where Communication Happens

Join At:  
- I/O (Network)  
- Memory  
- Processor

Program With:  
- Message Passing  
- Shared Memory  
- Dataflow/Systolic
Evolution of Architectural Models

Historically, machines tailored to programming models

- Programming model, communication abstraction, and machine organization lumped together as the “architecture”

Evolution helps understand convergence

- Identify core concepts

Most Common Models:

- Shared Address Space, Message Passing, Data Parallel

Other Models:

- Dataflow, Systolic Arrays

Examine programming model, motivation, intended applications, and contributions to convergence
Shared Address Space Architectures

Any processor can directly reference any memory location
- Communication occurs implicitly as result of loads and stores

Convenient:
- Location transparency
- Similar programming model to time-sharing on uniprocessors
  - Except processes run on different processors
  - Good throughput on multiprogrammed workloads

Naturally provided on wide range of platforms
- History dates at least to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

Popularly known as shared-memory machines / model
- Ambiguous: memory may be physically distributed among processors
Shared Address Space Model

**Process:** virtual address space plus one or more threads of control

Portions of address spaces of processes are shared

Virtual address spaces for a collection of processes communicating via shared addresses

Machine physical address space

- Writes to shared address visible to other threads, processes
- **Natural extension of uniprocessor model:** conventional memory operations for comm.; special atomic operations for synchronization
- OS uses shared memory to coordinate processes
Communication Hardware

Also a natural extension of a uniprocessor
Already have processor, one or more memory modules and I/O controllers connected by hardware interconnect of some sort

Memory capacity increased by adding modules, I/O by controllers ➔ Add processors for processing!
“Mainframe” approach:
- Motivated by multiprogramming
- Extends crossbar used for mem bw and I/O
- Originally processor cost limited to small scale
  - later, cost of crossbar
- Bandwidth scales with $P$
- High incremental cost; use multistage instead

“Minicomputer” approach:
- Almost all microprocessor systems have bus
- Motivated by multiprogramming, TP
- Used heavily for parallel computing
- Called symmetric multiprocessor (SMP)
- Latency larger than for uniprocessor
- Bus is bandwidth bottleneck
  - caching is key: coherence problem
- Low incremental cost
Example: Intel Pentium Pro Quad ('98)

- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth
Example: SUN Enterprise ('96)

- 16 cards of either type: processors + memory, or I/O
- Up to 30 processors
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus
Recent x86 Example ('17)

- Highly integrated, commodity systems
- On-chip: low-latency, high-bandwidth communication via shared cache
- Current scale = 4 processors (up to 12 on some models)
Scaling Up

• **Problem is interconnect**: cost (crossbar) or bandwidth (bus)
• **Dance-hall**: bandwidth still scalable, but lower cost than crossbar
  - latencies to memory uniform, but **uniformly large**
• **Distributed memory** or non-uniform memory access (**NUMA**)
  - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)
• **Caching shared** (particularly nonlocal) data?

“Dance hall”  Distributed memory
Example: Cray T3E ('96)

- Scale up to 1024 DEC ALPHA processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)
Example: SGI Altix UV 1000 (‘09)

- Scales up to 131,072 Xeon cores
- 15GB/sec links
- Hardware cache coherence for blocks of 16TB with 2,048 cores
Message Passing Architectures

Complete computer as building block, including I/O
  • Communication via explicit I/O operations

Programming model:
  • directly access only private address space (local memory)
  • communicate via explicit messages (send/receive)

High-level block diagram similar to distributed-mem SAS
  • But comm. integrated at IO level, need not put into memory system
  • Like networks of workstations (clusters), but tighter integration
  • Easier to build than scalable SAS

Programming model further from basic hardware ops
  • Library or OS intervention
Message Passing Abstraction

- **Send** specifies buffer to be transmitted and receiving process
- **Recv** specifies sending process and application storage to receive into
- **Memory to memory copy**, but need to name processes
- Optional tag on send and matching rule on receive
- User process names local data and entities in process/tag space too
- In simplest form, the send/recv match achieves pairwise synch event
  - Other variants too
- Many overheads: copying, buffer management, protection
Evolution of Message Passing

**Early machines: FIFO on each link**
- Hardware close to programming model
  - synchronous ops
- Replaced by DMA, enabling non-blocking ops
  - Buffered by system at destination until recv

**Diminishing role of topology**
- Store & forward routing: topology important
- Introduction of pipelined routing made it less so
- Cost is in node-network interface
- Simplifies programming
Example: Intel Paragon ('93)

Sandia’s Intel Paragon XP/S-based Supercomputer

2D grid network with processing node attached to every switch

Intel RISC processor (!!!)
Up to 2048 nodes
Example: IBM SP-2 (‘95)

- Made out of ~complete RS6000 (IBM RISC) workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)
Example: IBM Blue Gene/L ('04)

Nodes: 2 low-power PowerPC 400s
Everything but DRAM on-chip
Up to 64K nodes
Most power supercomputer for 3.5 years (until 2008)
First machine to achieve 100 TFLOPs on real application
Example: IBM Blue Gene/Q (‘11)

Successor to Blue Gene/L

Node: 18 cores, 4-way issue @ 1.6GHz, SIMD instructions, coherence within node

16 user cores (1 for OS, 1 spare)

Top of “green Top500” (2.1GFLOPS/W)
First to achieve 10PFLOPS on real application (100x BQ/L)
Taxonomy of Common Large-Scale SAS and MP Systems

aka “message passing”
Toward Architectural Convergence

Evolution and role of software have blurred boundary

- Send/recv supported on SAS machines via buffers
- Can construct global address space on MP using hashing
- Page-based (or finer-grained) shared virtual memory

Hardware organization converging too

- Tighter NI integration even for MP (low-latency, high-bandwidth)
- At lower level, even hardware SAS passes hardware messages

Even clusters of workstations/SMPs are parallel systems

- Emergence of fast system area networks (SAN)

Programming models distinct, but organizations converging

- Nodes connected by general network and communication assists
- Implementations also converging, at least in high-end machines
Convergence: General Parallel Architecture

A generic modern multiprocessor

Node: processor(s), memory system, plus communication assist
  • Network interface and communication controller
  • Scalable network
  • Convergence allows lots of innovation, now within framework
    • Integration of assist with node, what operations, how efficiently...
Intel Single-chip Cloud Computer (‘09)

48 cores
2D mesh network
- 24 tiles in 4x6 grid
- 2 cores / tile
- 16KB msg buffer / tile
4 DDR3 controllers

No hardware coherence
Message passing hardware
Coherence available through software library
Data Parallel Systems

Programming model:
- Operations performed in parallel on each element of data structure
- Logically single thread of control, performs sequential or parallel steps
- Conceptually, a processor associated with each data element

Architectural model:
- Array of many simple, cheap processors with little memory each
  - Processors don’t sequence through instructions
- Attached to a control processor that issues instructions
- Specialized and general communication, cheap global synchronization

Original motivation:
- Matches simple differential equation solvers
- Centralize high cost of instruction fetch & sequencing
Application of Data Parallelism

- Each PE contains an employee record with his/her salary
  
  If salary > 100K then
  
  salary = salary * 1.05

  else
  
  salary = salary * 1.10

- Logically, the whole operation is a single step
- Some processors enabled for arithmetic operation, others disabled

Other examples:
- Finite differences, linear algebra, ...
- Document searching, graphics, image processing, ...

Some examples:
- Thinking Machines CM-1, CM-2 (and CM-5)
- Maspar MP-1 and MP-2,
Evolution and Convergence

Rigid control structure (SIMD in Flynn taxonomy)

- SISD = uniprocessor, MIMD = multiprocessor

Popular when cost savings of centralized sequencer high

- 60s when CPU was a cabinet; replaced by vectors in mid-70s
- Revived in mid-80s when 32-bit datapath slices just fit on chip
- Caching, pipelining, and online scheduling weaken this argument

Other reasons for decline

- Simple, regular applications have good locality, can do well anyway
- Loss of applicability due to hardwiring data parallelism
  - MIMD machines as effective for data parallelism and more general

Programming model converges to SPMD (single program multiple data)

- Contributes need for fast global synchronization
- Structured global address space, implemented with either SAS or MP
Lasting Contributions of Data Parallel

“Multimedia extensions” of ISAs (e.g., SSE)
• Limited SIMD for 4-8 lanes

GPU computing
• Programming model looks like MIMD, but processor actually executes multi-threaded SIMD
• GPU jargon: vector lane == “core” ➜ 1000s of cores
• Reality: ~16 multithreaded SIMD cores
Example: Nvidia Pascal 100 (‘16)

60x streaming multiprocessors (SMs)
64 “CUDA cores” each
→ 3840 total “cores”

732 GB/s mem bw using 3D stacking technology

256KB registers / SM
Dataflow Architectures

Represent computation as a graph of essential dependences

- Logical processor at each node, activated by availability of operands
- Message (tokens) carrying tag of next instruction sent to next processor
- Tag compared with others in matching store; match fires execution
Evolution and Convergence

Key characteristics:
• Ability to name operations, synchronization, dynamic scheduling

Problems:
• Operations have locality across them, useful to group together
• Handling complex data structures like arrays
• Complexity of matching store and memory units
• Exposes too much parallelism (?)

Converged to use conventional processors and memory
• Support for large, dynamic set of threads to map to processors
• Typically shared address space as well
• But separation of programming model from hardware (like data parallel)

Lasting contributions:
• Out-of-order execution - mitigates problems for conventional, standard code
• Integration of communication with thread (handler) generation
• Tightly integrated communication and fine-grained synchronization
• Remained useful concept for software (compilers etc.)
Systolic/Spatial Architectures

- Replace single processor with **array of regular processing elements**
- **Orchestrate data flow** for high throughput with less memory access

Different from pipelining:
- Nonlinear array structure, multidirection data flow, each PE may have (small) local instruction and data memory

Different from SIMD: each PE may do something different

Initial motivation: VLSI enables inexpensive special-purpose chips

Represent algorithms directly by chips connected in regular pattern
Example: Systolic array for 1-D convolution

\[
y(i) = \sum_{j=1}^{k} w(j)x(i-j)
\]

- Practical realizations (e.g., iWARP) use quite general processors
  - Enable variety of algorithms on same hardware
- But dedicated interconnect channels
  - Data transfer directly from register to register across channel
- Specialized, and same problems as SIMD
  - General purpose systems work well for same algorithms (locality etc.)
- Recently, revived interest in neural network accelerators, processing-in-memory
MIT RAW Processor ('02)

Tiled mesh multicore
Very simple cores
No hardware coherence
Register-to-register messaging
Programmable routers

Programs split across cores
Looks like a systolic array!
Miscellany

No reading for today

Assignment #2 released!
- Due in two weeks
- 6 problems
- Please read Logistics section—different from #1!
- Please submit assignment as a group
Today

Wrap-up schools of parallel architecture

Measuring parallel performance

Amdahl’s Law and the limits of parallelism
Fundamental Design Issues
Sequential taxonomies not very useful
Programming models not enough alone, nor hardware structures
  • Same programming model can be implemented on different arch

Focus on Architectural distinctions that affect software

Need to design interface at several levels
  • Programmer sees: Programming model
  • Compiler / libraries / operating system
  • Software-to-hardware
Fundamental Design Issues

At any layer, interface (contract) aspect and performance aspects

- **Naming**: How are logically shared data and/or processes referenced?
- **Operations**: What operations are provided on these data
- **Ordering**: How are accesses to data ordered and coordinated?

Understand at programming model first, since that sets requirements

Other design & implementation issues:

- **Node Granularity**: How to split between processors and memory?
- **Communication Cost**: Latency, bandwidth, overhead, occupancy...
- **Replication**: How are data replicated to reduce communication?
Sequential Programming Model

**Naming:**

*Can name any variable in virtual address space*

- Hardware/compilers translate to physical addresses

**Operations:**

*Loads and Stores*

**Ordering:**

*Sequential program order*

- E.g., read-after-write (RAW) or “true” dependences
- Also: write-after-read (WAR) or “anti-“ dependences, write-after-write (WAW) or “output” dependences, and “control” dependences
Sequential Performance

Compilers and hardware violate other orders without getting caught
• Compiler: reordering and register allocation
• Hardware: out of order, speculation

Modern, OOO cores eliminate all but true dependences

Communication costs: Accessing memory (or L2 cache)
Replication: Transparent replication in caches (in hw)
SAS Programming Model

Naming:

- Any process can name any variable in shared space

Operations:

- Loads and stores, plus those needed for ordering

Simplest Ordering Model: Sequential Consistency (SC)

- Within a process/thread: sequential program order
- Across threads: some interleaving (as in time-sharing)
- Additional orders through synchronization
- Again, compilers/hardware can violate orders without getting caught

Today, machines implement weaker consistency models than SC to relax ordering restrictions and increase parallelism (discussed later)
Add'l Ordering via Synchronization

**Mutual exclusion (locks)**

- Intuition: Room that only one person can enter at a time
- `acquire()`/`release()` – only one thread at a time can acquire
- Ensure certain operations on certain data can be performed by only one thread at a time
- No ordering guarantees among threads
- Example: transferring money between bank accounts

**Many other ways to synchronize**

- **Events** (“wait until X is set by someone else”)
- **Barriers** (“wait until X threads reach here”)
- **Read-write locks** (allow many readers, but only one writer)
- **Semaphores** (allow up to X concurrent threads)
- Many more...
Message Passing Programming Model

Naming:
- Local memory
- Remote processes (+ tag)
  - No global addressing!

Operations:
- Load/store locally
- Send/receive remote

Ordering:
- Program order locally
- Messages order sender-receiver remote
- Mutual exclusion for free (in pure message passing)
## Comparison of Parallel Arch Schools

<table>
<thead>
<tr>
<th>School</th>
<th>Naming</th>
<th>Operations</th>
<th>Ordering</th>
<th>Processing Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Anything</td>
<td>Load/store</td>
<td>Program</td>
<td>Large (ILP)</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Anything</td>
<td>Load/store</td>
<td>SC + synch</td>
<td>Large-to-medium</td>
</tr>
<tr>
<td>Message passing</td>
<td>Remote processes</td>
<td>Send/receive</td>
<td>Messages</td>
<td>Large-to-medium</td>
</tr>
<tr>
<td>Dataflow</td>
<td>Operations</td>
<td>Send token</td>
<td>Tokens</td>
<td>Small</td>
</tr>
<tr>
<td>Data parallel</td>
<td>Anything</td>
<td>Simple compute</td>
<td>Bulk-parallel</td>
<td>Tiny</td>
</tr>
<tr>
<td>Systolic/spatial</td>
<td>Local mem + input</td>
<td>Complex compute</td>
<td>Local messages</td>
<td>Small</td>
</tr>
</tbody>
</table>
Design Issues Apply at All Layers

Programming model’s position provides constraints/goals for system

In fact, each interface between layers supports or takes a position on:

- Naming model
- Set of operations on names
- Ordering model
- Replication
- Communication performance

Any set of positions can be mapped to any other by software

Let's see issues across layers:

- How lower layers can support contracts of programming models
- Performance issues
Same Model, Diff Implementation

Example: Shared address space in programming model

Supported in hardware
Hardware handles transfers and bookkeeping (eg, consistency)

Supported in software
- Can provide SAS through OS using virtual memory
  - page mappings only for data that are local
  - remote data accesses incur page faults; brought in via page fault handlers
  - same programming model, different hardware reqs and cost model
- Or through compilers or runtime
  - program tags shared objects
  - compiler instruments accesses to shared objects
  - can be more efficient than OS (eg, app knows how big each object is, sharing pattern, etc)
Same Model, Diff Implementation

Example: Implementing Message Passing

Support at hardware interface
  - Limited flexibility for different apps (matching tags, allocating buffers)

Support at system/user interface in software (most common)
  - Hardware interface provides basic data transport (well suited)
  - Choices at user/system interface:
    - OS only...but syscalls are expensive
    - OS sets up protection, userspace software handles buffers directly
  - Or, if hardware supports SAS, just use load/stores (allocate buffer in destination local memory)

Need to examine the issues and tradeoffs at every layer
  - Frequencies and types of operations, costs
Communication Performance

Performance characteristics determine usage of operations at a layer

- Programmer, compilers etc make choices based on this

Fundamentally, three characteristics:

- **Latency**: time taken for an operation
- **Bandwidth**: rate of performing operations
- **Cost**: impact on execution time of program

If processor does one thing at a time:

\[ \text{bandwidth} \propto \frac{1}{\text{latency}} \]

- But actually more complex in modern systems
Communication Cost Model

Communication Time per Message

\[ \text{Communication Time per Message} = \text{Overhead} + \text{Assist Occupancy} + \text{Network Delay} + \frac{\text{Size}}{\text{Bandwidth}} + \text{Contention} \]

\[ = o_v + o_c + l + \frac{n}{B} + T_c \]

**Overhead and assist occupancy** may be \( O(n) \) or not

Each component along the way has occupancy and delay
- Overall delay is sum of delays
- Overall occupancy (1/bandwidth) is biggest of occupancies

**Total Comm. Cost**

\[ \text{Total Comm. Cost} = \text{frequency} \times \left( \text{Comm. time} - \text{overlap} \right) \]
Replication

Very important for reducing data transfer/communication
Again, depends on naming model

Uniprocessor: caches do it automatically
  • Reduce communication with memory

Message Passing naming model at an interface
  • A receive replicates, giving a new name; subsequently use new name
  • Replication is explicit in software above that interface

SAS naming model at an interface
  • A load brings in data transparently, so can replicate transparently
  • Hardware caches do this, e.g. in shared physical address space
  • OS can do it at page level in shared virtual address space, or objects
  • No explicit renaming, many copies for same name: coherence problem
    - in uniprocessors, “coherence” of copies is natural in memory hierarchy
Are We Asking Right Questions?

• Programming model:
  • SAS/MP/DP?
  • Is this what should be exposed to the programmer?

• Design issues:
  • Naming/operations/ordering/replication/communication
  • Should any of this be exposed to programmer?

Holy grail is a system that
• Is easy to program
• Yields good performance (and efficiency)
• Scales well (adding more resources improves performance)

Can these be achieved with evolutionary improvements?
Or do we need to rethink computing for parallelism?
Recap

Exotic designs have contributed much, but given way to convergence
- Push of technology, cost and application performance
- Basic processor-memory architecture is the same
- Key architectural issue is in communication architecture

Fundamental design issues:
- Functional: naming, operations, ordering
- Performance: organization, replication, performance characteristics

Design decisions driven by workload-driven evaluation
- Integral part of the engineering focus

MAKE THE COMMON CASE FAST
Performance Metrics
Parallel Speedup

Time to execute the program with 1 processor divided by
Time to execute the program with N processors
Parallel Speedup Example

\[ a4x^4 + a3x^3 + a2x^2 + a1x + a0 \]

Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

How fast is this with 3 processors?
Takeaway

To calculate parallel speedup fairly you need to use the best known algorithm for each system with \( N \) processors.

If not, you can get \textit{superlinear speedup}.
Superlinear Speedup

Can speedup be greater than P with P processing elements?

Consider:
- Cache effects
- Memory effects
- Working set

Happens in two ways:
- Unfair comparisons
- Memory effects
Utilization, Redundancy, Efficiency

Traditional metrics
  · Assume all P processors are tied up for parallel computation

Utilization: How much processing capability is used
  · $U = \frac{\text{(# Operations in parallel version)}}{\text{(processors} \times \text{Time)}}$

Redundancy: how much extra work is done
  · $R = \frac{\text{(# of operations in parallel version)}}{\text{(# operations in best uni-processor algorithm version)}}$

Efficiency
  · $E = \frac{\text{(Time with 1 processor)}}{\text{(processors} \times \text{Time with P procs)}}$
  · $E = \frac{U}{R}$
AMDAHL’S LAW: 
THE LIMITS OF PARALLELISM
Amdahl's law

You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

<table>
<thead>
<tr>
<th></th>
<th>total trip time</th>
<th>speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>16.5 hours</td>
<td>1</td>
</tr>
<tr>
<td>SST</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!
**Amdahl's law (cont)**

Old program (unenhanced)

\[
\begin{array}{c|c}
T_1 & T_2 \\
\end{array}
\]

Old time: \( T = T_1 + T_2 \)

New program (enhanced)

\[
\begin{array}{c|c}
T_1' = T_1 & T_2' \leq T_2 \\
\end{array}
\]

New time: \( T' = T_1' + T_2' \)

Speedup: \( S_{\text{overall}} = \frac{T}{T'} \)

\( T_1 \) = time that can NOT be enhanced.

\( T_2 \) = time that can be enhanced.

\( T_2' \) = time after the enhancement.
Amdahl’s law (cont)

Two key parameters:

\[ F_{\text{enhanced}} = \frac{T_2}{T} \quad \text{(fraction of original time that can be improved)} \]
\[ S_{\text{enhanced}} = \frac{T_2}{T_2'} \quad \text{(speedup of enhanced part)} \]

Amdahl’s Law:

\[ S_{\text{overall}} = \frac{T}{T'} = \frac{1}{(1 - F_{\text{enhanced}}) + \frac{F_{\text{enhanced}}}{S_{\text{enhanced}}}} \]

Key idea: Amdahl’s law quantifies the general notion of diminishing returns. It applies to any activity, not just computer programs.
Amdahl's law (cont)

Trip example: Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

<table>
<thead>
<tr>
<th></th>
<th>time NY-&gt;Paris</th>
<th>total trip time</th>
<th>speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>8.5 hours</td>
<td>16.5 hours</td>
<td>1</td>
</tr>
<tr>
<td>SST</td>
<td>3.75 hours</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
<tr>
<td>rocket</td>
<td>0.25 hours</td>
<td>8.25 hours</td>
<td>2.0</td>
</tr>
<tr>
<td>rip</td>
<td>0.0 hours</td>
<td>8 hours</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Amdahl's law (cont)

Corollary: \( 1 \leq S_{overall} \leq \frac{1}{1 - F_{enhanced}} \)

<table>
<thead>
<tr>
<th>( F_{enhanced} )</th>
<th>( \text{Max } S_{overall} )</th>
<th>( F_{enhanced} )</th>
<th>( \text{Max } S_{overall} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>0.9375</td>
<td>16</td>
</tr>
<tr>
<td>0.5</td>
<td>2</td>
<td>0.96875</td>
<td>32</td>
</tr>
<tr>
<td>0.75</td>
<td>4</td>
<td>0.984375</td>
<td>64</td>
</tr>
<tr>
<td>0.875</td>
<td>8</td>
<td>0.9921875</td>
<td>128</td>
</tr>
</tbody>
</table>

*Moral*: It is hard to speed up a program.

*Moral++*: It is easy to make premature optimizations.
Caveats of Parallelism (I): Amdahl’s Law

Amdahl’s Law

- $f$: Parallelizable fraction of a program
- $P$: Number of processors

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}
\]


Maximum speedup limited by serial portion: Serial bottleneck
Sequential Bottleneck

![Graph showing Sequential Bottleneck with Speedup on the y-axis and f (parallel fraction) on the x-axis. The graph includes lines for N=10, N=100, and N=1000.](image)
Why the Sequential Bottleneck?

Parallel machines have the sequential bottleneck

Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

for ( i = 0 ; i < N; i++)

Single thread prepares data and spawns parallel tasks (usually sequential)
Implications of Amdahl’s Law on Design

- CRAY-1

- Well known as a fast vector machine
  - 8 64-element vector registers

- The fastest SCALAR machine of its time!
  - Reason: Sequential bottleneck!
Caveats of Parallelism (II)

Amdahl’s Law

• $f$: Parallelizable fraction of a program
• $P$: Number of processors

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}
\]

• Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.

Maximum speedup limited by serial portion: Serial bottleneck

Parallel portion is usually not perfectly efficient

• Synchronization overhead (e.g., updates to shared data)
• Load imbalance overhead (imperfect parallelization)
• Resource sharing overhead (contention among $N$ processors)
Bottlenecks in Parallel Portion

**Synchronization:** Operations manipulating shared data cannot be parallelized
- Locks, mutual exclusion, barrier synchronization
- *Communication:* Tasks may need values from each other
  - Causes thread serialization when shared data is contended

**Load Imbalance:** Parallel tasks may have different lengths
- Due to imperfect parallelization or microarchitectural effects
  - Reduces speedup in parallel portion

**Resource Contention:** Parallel tasks can share hardware resources, delaying each other
- Replicating all resources (e.g., memory) expensive
  - Additional latency not present when each task runs alone
Difficulty in Parallel Programming

Little difficulty if parallelism is natural
- “Embarrassingly parallel” applications
- Multimedia, physical simulation, graphics
- Large web servers, databases?

Big difficulty is in
- Harder to parallelize algorithms
- Getting parallel programs to work correctly
- Optimizing performance in the presence of bottlenecks

Much of parallel computer architecture is about
- Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
- Making programmer’s job easier in writing correct and high-performance parallel programs